2

CDRL 01270
TASK BR67

The BOEING Company
D613-21270

CDRL 01270

Software Technology for Adaptable, Reliable Systems (STARS)

Submitted to:
Electronic Systems Division:
Air Force Systems Command, USAF:
Hanscom AFB, MA 01731-5000:

Contract No:
F19628-88-D-0028

CDRL 01270
Inter-Tool Communication Facility (ITCF) Final Report

February 16, 1990

Prepared by:
SofTech, Inc
San Diego, California

Submitted by:
The Boeing Company
Defense and Space Group
Systems and Software Engineering
P.O. Box 3999
Seattle, Washington 98124

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188.*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE 16-FEB-90	3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE Inter-Tool Communication Facility (ITCF) Final Report	5. FUNDING NUMBERS C: F19628-88-D-0028 TA: BR-67
6. AUTHOR(S) Carl Hitchon, SofTech Inc.	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) The Boeing Company Boeing Aerospace and Electronics Division Systems and Software Engineering P.O. Box 3999 Seattle, Washington 98124	8. PERFORMING ORGANIZATION REPORT NUMBER D-613-21270
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) ESD/AVS Bldg. 17-04 Room 113 Hanscom Air Force Base, 01731-5000	10. SPONSORING / MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release - distribution unlimited.	12b. DISTRIBUTION CODE A

13. ABSTRACT *(Maximum 200 words)*

The Inter-Tool Communication Facility (ITCF) was specified under STARS Task Q11. A VAX/CAIS-A implementation of the core ITCF facilites, a demonstration of the ITCF using existing tools, and this final report were completed under STARS Task BR67. This report describes the ITCF, the design, the implementation, and the demonstration performed. More detailed documentation and information about the ITCF and the demonstration tools can be found in the ITCF Version Description and in the source code for the ITCF packages and for demonstration tools. All ITCF deliverables, incl. ling source code and reports are currently on the Boeing MDSC VAX under the root directory.

14. SUBJECT TERMS Keywords: STARS ITCF	15. NUMBER OF PAGES 32
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT None

Inter-Tool Communication Facility (ITCF) Final Report

Prepared by

John Neorr

for Carl Hitchon, SofTech Inc. 2/16/90
Chief Programmer (BR67)
Inter-Tool Communication

Reviewed by

J E King

James E. King
System Architect

Reviewed by

John Neorr

John M. Neorr 2/16/90
Development Manager

Approved by

W M. Hodges 2/16/90

William M. Hodges
STARS Program Manager

TABLE OF CONTENTS

TABLE OF CONTENTS [continued]

1. INTRODUCTION

The Inter-Tool Communication Facility (ITCF) was specified under STARS Task Q11. A VAX/CAIS-A implementation of the core ITCF facilities, a demonstration of the ITCF using existing tools, and this final report were completed under STARS Task BR67. This reports describes the ITCF, the design, the implementation, and the demonstration performed. More detailed documentation and information about the ITCF and the demonstration tools can be found in the ITCF Version Description and in the source code for the ITCF packages and for demonstration tools. All ITCF deliverables including source code and reports are currently on the Boeing MDSC VAX under the root directory:

SYS$USER7[SOFTECH.USERS.SABBUHL.ITCF]

The file contained therein describing the directory organization of the deliverables is:

[SOFTECH.USERS.SABBUHL.ITCF]ITCF_VERSION_DESCRIPTION.ADS.

2. GENERAL DESCRIPTION

The following is a general description of the ITCF. More details can be found in the file ITCF_Version_Description.ads and in the package specifications and bodies for the delivered software.

The ITCF permits concurrently executing tools to cooperate in an integrated manner while maintaining their high degree of modularity and functional independence. The ITCF offers a layered architecture of communication services which may be adapted to meet the requirements of a variety of tool integration strategies. The principle conceptual layers are streams, stream protocols and messages.

The ITCF services manage the complex details of inter-tool communication and provide conceptually simple and easy to use interfaces to the tool writer. These services facilitate the integration of tools with relatively modest effort, including existing tools which were not originally designed to use the ITCF packages.

The ITCF is intended to transfer data which is primarily transient in nature as opposed to data which is permanently stored in the environment data base. Permanent data should be handled through the environment object management system. Nevertheless, simple tools can be implemented, using the ITCF packages, which allow the connection of other tools to files (including devices) as sources and sinks for inter-tool data.

The ITCF is also intended to support communication between tools executing in different CAIS-A environments that are connected by a gateway. However, this feature was not implemented under task BR67.

3. ITCF CONCEPTS

The underlying concepts of the ITCF have been derived from a number of sources. These include UNIX pipes, plumbing fixtures and filters; the CAIS-A IO Connection Model; and various Remote Procedure Call (RPC) facilities and stream manipulation packages. The FIELD environment integration mechanism, described by Steven Reiss of Brown University, was particularly influential.

3.1 Clients and Connection Centers

The focus of ITCF operations are server processes called connection centers. A given connection center may have one or more clients. A client may be a tool (i.e. a process), or a gateway. A gateway client represents other clients and/or connection centers on a remote host system. Clients communicate with one another via the connection center.

The first step in using the ITCF is the creation of a connection center. After the center has been created, clients may be connected to the center. Connection of a client to a connection center establishes a communications channel between the client and the center. This channel is called a connection. Clients may be connected or disconnected at any time. Only clients of the same center may communicate with one another. Each client may be connected to only one center at a given time. Clients may communicate through the connection center concurrently.

3.2 Streams

At the lowest conceptual layer of the ITCF, data is passed through the connection center in the form of streams. A stream is simply a sequence of data (of indefinite length) generated by a client. A client sends data by writing into a stream. A client receives data by reading from a stream.

The flow of data in a stream is uni-directional. When bi-directional flow is required, a reverse stream called a reply stream may be used. A client process may use multiple streams concurrently. The flow of data in each stream is independent of other unrelated streams.

Streams may be used by processes for direct communication without the use of a connection center. Connection center facilities are built on top of the lower-level stream services. Streams may also be used locally within a process for communication between tasks without the overhead associated with sending streams through a separate connection center process.

4. Implementation Overview

The core facilities of the ITCF were implemented on the MDSC VAX machine using VAXAda 2.0 and Version 4.5 of the CAIS-A implementation. The ITCF services are provided as a collection of packages containing interfaces which may be called by CAIS tools.

The principal packages implemented are:

Stream_IO - Supports all low-level stream operations including interprocess communication.

Text_Stream_IO - Provides the interfaces of Ada Text_IO for reading and writing stream data.

Connection - Provides interfaces that client tools call to interface with a connection center.

Connection_Center - A generic package which supports the instantiation of user defined connection centers.

Distribution - A generic package which performs general distribution list management functions for instantiations of package Connection_Center.

Monitor - A low-level packages which is used to manage concurrent access to streams. This is an internal ITCF package which contains no tool interfaces.

5. DESIGN PROBLEMS AND SOLUTIONS

Aspects of the ITCF implementation which presented significant technical challenges were the management of concurrent access to streams and the interprocess connection resources, and the mechanism to support stream copying. Stream copying is a fundamental ITCF concept of substantial power in affecting the distribution of streams and the composition of streams from other streams.

5.1 Concurrent Access to Streams

Two problems needed to be addressed in the handling of concurrent access. The first was to develop a conceptual model that would make a correct implementation conceptually manageable. The second problem was to assure that management of concurrent access would be reasonably efficient.

5.1.1 Using Ada Rendezvous

Initial attempts to achieve a conceptual model for concurrent access were focused on the direct use of the Ada rendezvous model. One question was how tasking should be applied. The Ada rendezvous model is generally applied by identifying a object to which concurrent access may occur and then applying a task to manage the object. All access to the object is then accomplished through calling entries to that task. Hence, one attempt to the solve concurrent access problems in the ITCF was to have each stream be managed by a separate task. This approach was eventually rejected because it would require a large number of tasks, straining system resources. Also, it was not clear how stream copying could be implemented. Stream copying would either require 1) stream-tasks to call other stream-tasks, thereby making themselves unavailable for satisfying other concurrent access to the stream, or 2) the creation of addition copy-tasks that would simply read data from one stream-task and write-data to another. While such tasks could be recycled, thus reducing the overall overhead of frequent task creation, the number of tasks that would be required was still of some concern.

Another approach to using rendezvous was to define a single task which provides all services to all streams. This task would serialize access to all stream operations. While this approach would greatly reduce the number of tasks required, it had some disadvantages. One disadvantage was that all stream functions would have to be provided in a single large task body. In order to modularize these functions, calls would have to be used in the ACCEPT statements of this task body adding additional overhead (which could possibly be reduced through the use of PRAGMA INLINE). Also, how to interface with other tasks which manage transmission of streams over connections was not clear.

While these problems probably could be overcome, there was one problem that was shared by all approaches that used Ada rendezvous exclusively: every operation on a stream would involve four tasks switches. Task switching has certain unavoidable overhead even in the best Ada runtime implementations. All registers must be stored and loaded, and the scheduling algorithm must be invoked. This would imply that an operation which is to write a single byte to a stream could have an overhead on the order of a hundred instructions. This overhead could be significantly reduced by using buffering to minimize the number of rendezvous required to transmit a given amount of data. However, this would only help in the Put and Get functions in package Stream_IO. The overhead for all other functions would not be reduced.

5.1.2 Using Monitors

An alternative approach was to apply Hoare's Monitor concept. In this approach, the set of all streams is considered a shared resource to which access is control via a monitor entity. Whenever a task needs to operate on a stream, it must first enter the monitor. Monitor entry is queued so that only one task can execute within the monitor at a time. If the operation cannot be completed immediately (for example an attempt to read a stream which is temporarily empty), the task may suspend itself on a condition variable until the operation can be completed. When a task suspends itself within a monitor, another task is allowed to enter. Eventually, a task will enter which writes to the stream and signals the condition variable upon which the suspended task is waiting.

5.1.3 Monitor Implementation

The monitor framework was very helpful in sorting out the concurrent access problems. However, since Ada does not directly support monitors, it was necessary to consider how they would be implemented. Waiting must occur when a task attempts to enter a monitor, if another task is using the monitor, and when a task must be suspended on a condition variable. In Ada, the only way for a task to wait until some event occurs is through a call to a task entry. A very simple and direct approach to the monitor entry problem is to use a simple task which alternately accepts an entry call to enter the monitor and an entry call to leave the monitor. The major drawback to this solution is that the cycle of entering and leaving the monitor would require two rendezvous involving eight task switches.

However, an important observation is that, in most cases, when a task attempts to enter the monitor there will be no task executing in the monitor. Thus, in most cases a rendezvous should be unnecessary. In the ITCF implementation, a counter which can be incremented/decremented and tested as an atomic operation is used to determine whether a task is currently executing within the monitor. A rendezvous is used to suspend an entering task only if the result of incrementing the counter indicates that another task is already executing inside the monitor. This strategy reduces the overhead of monitor entry and exit to a few instructions, versus the hundreds of instructions that would be executed using rendezvous exclusively.

The technique used could be criticized for not being pure Ada. However, the only implementation dependency introduced by this technique is the implementation of the atomic counter on the host. Most processors have such an instruction. Others have an instruction (such as a test and set) which can be used to synthesize an atomic counter. In the case of the VAX, the VAXAda compiler has a built-in procedure for an atomic counter. For other hosts or compiling systems, it may be necessary to write a small assembly language unit.

In any case, the dependency is isolated within a small package called Protected_Counter. There is no dependency on the Ada run-time implementation. All tasking operations are still handled in pure Ada.

5.2 Stream Copying

Several false starts were required in the design of stream copying before a satisfactory solution was arrived at. The final solution was to place an item representing the "tail" (i.e. the remainder) of a copied stream in the item queue of every target stream to which the stream is copied. These tails are kept on a list. Each time another data buffer is added to the source stream, an item referencing that buffer is inserted into each target stream immediately ahead of each tail (in the list of tails) for the copied stream. This effectively copies the data into each target stream at the appropriate point within each of those streams. When the copied stream is closed, the tails are discarded since no further insertions of data from the copied stream are necessary.

In applications involving large streams, the amount of data which is buffered in a stream must be bounded to prevent storage resources from being exhausted. Handling of bounded buffering in streams was problematic due to the copying facility. Initial, it was not clear how to account for buffered data which has been copied. As the design of the copy capability progressed, it was decided that buffers would be copied by reference rather than by value for efficiency reasons. The accounting problem of placing bounds on buffering was then solved by designating the stream to which data is originally written as the owner of that data. Thus, copying a stream A to a stream B has no affect on the amount of data that can be written directly to stream B. A shared buffer is only released after every reference to it in any stream has been deleted. Thus, write operations to stream A which has been copied to stream B may be blocked until data is read from both streams A and stream B.

6. CHANGES TO THE ORIGINAL DESIGN

Few changes were made to the ITCF specifications during the implementation. However, one significant change was a redistribution of the functionality provided by the packages Stream_IO and Stream_Services. During the implementation it was realized that the underlying stream services (which were not detailed in the specification) could be useful outside the context of ITCF connection centers. Thus, interfaces originally intended to be hidden in the implementation of Stream_Services were instead made a visible part of package Stream_IO. Thus, package Stream_IO now provides full access to low-level stream functionality. Package Stream_Services was eliminated.

Higher-level functionality, associated specifically with the use of a connection center, was removed from package Stream_IO and placed in package Connection which already included interfaces through which client processes communicate with a connection center. (One regret here is the package name Connection which would more appropriately be Center_IO. This could be fixed in a future version.) The rearrangement of these services improves ITCF modularity since general-purpose stream services (provided by Stream_IO) can now be used without the overhead of also binding-in connection center support.

7. DEMONSTRATION OF THE ITCF

The selection of a demonstration for the ITCF was a compromise between the desire to demonstrate a meaningful, non-trivial ITCF application and the need to devote most of the available time to the implementation of the ITCF. A meaningful demonstration would be one in which the functionality of the existing tools is integrated. The repository containing STARS and SIMTEL tools was considered as the primary source of existing tools. In order to minimize the risk of demonstration problems, attention was focused on relatively simple, small tools. A demonstration of the ability to integrate tools was of more interest than the tools themselves. For example, integration of the ITCF with ACE (Ada Command Environment) was considered but was not undertaken, primarily because ACE is large body of software which would have had to be ported from a UNIX environment to CAIS-A on a VAX. Also, the integration of ITCF with ACE would not in itself demonstrate the utility of the ITCF. At least two more tools would be required to perform some sort of demonstration.

7.1 Tool Selection

There were several tools in the repository which could perform operations on Ada source code. These included editors, spelling checkers, Ada statement counters and Ada source code reformatters. Since programmer's spend much of their time using text editors, it seemed that a natural choice for a demonstration would be one which extended the functionality of a text editor through integration with other text processing tools. Thus, a demonstration was conceived in which an editor would be the primary tool interfacing with the user and would be integrated with other text processing tools operating as servers through a connection center. This scenario seemed to provide a relatively straight-forward demonstration.

A simple line editor, written in Ada, was chosen from the repository. Although it is unlikely that this particular editor would be a heavily used tool in an actual SEE, the same types of extensions could be made to the more popular varieties of screen editors. For purposes of the demonstration, additional commands were added to the editor which would be carried out through a connection center. In particular, new commands to reformat and to count statements in Ada source code being edited were added. These new commands cause the contents of the current editor buffer to be transmitted to other tools via a connection center and a reply to be received by the editor.

An Ada source code reformatter was selected from the repository. There were at least two choices. The simpler and smaller reformatter was chosen. Initially, a spelling checker was also selected as a server tool for the demonstration. However, as time began to run out it was decided to replace the spelling checker with a simpler tool, an Ada statement counter, to reduce the risk of not meeting schedules. The main reasons for the substitution was that the spelling checking had a very long startup time and its user interface seemed complex. It was not clear how much effort would be required to adapt it for an acceptable demonstration.

In order to demonstrate the broadcast and reply concatenation capabilities of the ITCF, another simple command was added to the line editor. This command causes a message to be sent to all clients of a connection center which have registered themselves as server tools. Each tool responds with its registered name. These responses are concatenated into a single stream by the connection center and delivered back to the editor which prints the response. Hence, the net affect of the this new editor command is to poll for the names of all available server tools which are connected to the center.

7.2 Tool Adaptation

Adaptation of the tools chosen for integration via the ITCF was straight-forward. In the case of the line editor, the new commands were readily added. However, the command for reformatting suggested that another editor extension would be necessary. The problem is that if there are mistakes in the source code, the results of reformatting could be erroneous in which case the user may wish to restore the prior contents of the editor buffer in place of the reformatted contents. The mechanism used by the editor to store the text being edited was easily modified to handle two text buffers instead of one. Another command was added to the editor which allows the user to toggle between these buffers. Thus, after a buffer is formatted the user has the option of switching back to the original text buffer as it was before invoking the reformatter.

Both the reformatter and the Ada statement counter tools were easily adapted since they already used TEXT_IO for purposes of input and output. Adaptation of these tools was largely a matter of adding a few calls to the ITCF package Connection. Each tool had to perform the following initialization steps:

1. Call Open_Connection to initialize the connection to the connection center.

2. Call Register_Pattern to register the identification of the tool with the connection center.

3. Call Activate_Connection to notify the connection center that the tool was ready to receive request streams.

Thereafter, the tools would call the Receive interface in package Connection to receive each stream containing source code to be processed. The tools would then open each stream using TEXT_STREAM_IO. Since these tools already used TEXT_IO to read their input and write their output, the TEXT_IO calls had only to be changed to call TEXT_STREAM_IO instead.

More detailed information about the changes made to tools for adaptation can be found in the Ada source code files for the tools

7.3 Tool Adaptation Problems

Although the changes necessary to integrate the tools with the ITCF were minor, other aspects of tool behavior became important in the context of integration. This was particularly evident in the reformatter tool. The reformatter, as it was written, would stop processing if it encountered anything which is not correct Ada in its input stream. While this sort of behavior may be acceptable in a stand alone tool, in an integration setting it seemed counter-productive. If the user sent his editor buffer to the reformatter, only part would come back in the (very likely) event that the source code contained errors. Hence, changes were made to the reformatter to cause it to pass remaining source code through unchanged in the event of a syntactic error from which it could not recover.

The reuse of tools such as the reformatter in this way increases the need for tool reliability. While occasional failures of a stand-alone tool may be tolerable, the failure of a tool in a closely integrated scenario may not be so well tolerated. For example, an obvious and implied extension of the demonstration scenario would be the multiplexed use of the connection center and the server tools (the reformatter and the statement counter) by several independent invocations of the editor tool. That is, the services provided by the connection center could be provided to multiple independent users. In this case, if one user sends an source program to the reformatter which causes it to abort, then the facility is also lost to the other users of the connection center.

Hence, the robustness of tools becomes more critical. It may be that future adaptations of tools to the ITCF should include considerations such as restarting a tool if it fails. At the very least, when one tool fails, other tools connected to connection center should not necessarily fail as a consequence.

7.4 Tool Adaptation Effort

Much of the effort that was necessary to adapt the reformatter was spent in trying to fix existing weaknesses and make it more robust. There are still existing bugs in this tool which could not be fully investigated in the time available. Approximately a two man-weeks of effort were applied to this tool.

Adaptation of the Ada statement counter was trivial. requiring only a few hours. It was only necessary to add a main processing loop.

Adaptation of the line editor was fairly straight-forward. Some knowledge of its internal structure was necessary in order to add the new commands and to add the capability of multiple buffers which could be swapped via a command. The editor adaptations required approximately a man-week.

Note that all adaptations were performed by a programmer familiar with the ITCF interfaces but completely unfamiliar with the tools being integrated. No consultation from others was sought in performing the adaptations. Realistically, this situation (no consultation) probably reflects typical future tool adaptation scenarios. However, those with expertise in these tools could have performed the necessary adaptations more quickly.

7.5 Instantiation of the Demonstration Connection Center

For flexibility, ITCF connection center software is provided in the form of generic packages which must be instantiated by the user to create a connection center having the desired distribution strategy. The connection center instantiations used in the demonstration are in a package called Integration_Center. The actual parameters for the instantiation include an Ada type defining patterns and a matching function which compares streams to be distributed against patterns. The Integration_Center package defines patterns to be simple strings up to 20 characters long. These strings are the names of the server tools connected to the Integration_Center, e.g. "Formatter" and "Ada Count". Requests sent by the line editor to the connection center have a header which indicates which tool is to receive the stream. A stream with the header "*" is sent to all of the server tools connected. The "*" header is used to implement the editor command which polls the server tools connected to the connection center.

The implementation of the Integration_Center was relatively simple since most of the work is already taken care of in the of bodies of the generic packages. However, instantiation of package Distribution is non-trivial, requiring a detailed understanding of how to distribute call streams and how to concatenate their corresponding reply streams. The process of making a connection center could be made substantially easier by providing a higher-level generic package which would automatically instantiate the package Distribution with a standard distribution strategy.

7.6 Demonstration Performance Results

Responsiveness of the special editor commands was found to be fairly good once all tools and the connection center were established and initialized. Typical response times for small editor buffers is one or two seconds. However, the initial establishment of the connection center and the tools is fairly lengthy primarily due to the many node creation operations performed by the CAIS-A implementation in creating the processes and IO channels (six nodes are created for the demonstration configuration). It is expected that these creation times will be much shorter in future. improved releases of the CAIS-A implementation.

Performance was significantly poorer when large streams were transmitted. A three page source file can take on the order of 20 seconds to be reformatted. The computational overhead for stream management is relatively low. The apparent cause of the performance problem with large streams is the use of polling in the low-level IO mechanism. It is likely that performance would be much better if this polling mechanism were replaced with an interrupt-driven IO mechanism.

7.7 Running the Demonstration

The following steps may be used to run the demonstration on the Boeing MDSC VAX:

1. Logon as user SOFTECH1, password GREENE.

2. Type: RUN UOBJ:CAIS_LOGON to logon to the CAIS. Specify user ITCF_DEMO, password ITCF_DEMO.

3. Wait for the prompt "file?". This may take a minute.

4. Type the name of a VMS file to read into the buffer;

   or,

5. Type a carriage return to start an empty buffer.

6. Type "H" (help) and "S" (summary) for an editor command summary.

7. Type "A" (append) to enter an Ada program from the keyboard. Type "." CR to end input mode.

8. Type "L" (list) "A" (all) to list the editor buffer contents.

9. Type "C" (center) to poll the tools connected to the ITCF connection center. The names of the tools will be printed.

10. Type "Z" to send the editor buffer to the Formatter tool. A new buffer will be returned.

11. Type "B" (buffer switch) to toggle back to the unformatted buffer.

12. Type "K" to send the editor buffer to the Ada statement counter. The number of Ada statements, blank lines, comment lines etc. will be printed.

13. If CONTROL-Y must be typed, refresh the database before logging into the CAIS again, as follows:

    Type CONTROL-Y
    Type STOP
    Type @REFRESH

8. UNIMPLEMENTED FEATURES

Not all features of the ITCF were implemented under this task. Only the core facilities required for the demonstration were fully implemented. In particular, the following ITCF packages were not implemented:

> **Package Binary_Stream_IO** – protocol for transmitting Ada data structures in a compact binary form.
>
> **Package Sequential_Stream_IO** – an ITCF version of Ada's Sequential_IO.
>
> **Package Message_IO** – a generic package for the convenient encoding and decoding of short streams.

Also, the capability to connect together several connection centers was not implemented. This feature is useful primarily in the implementation connections between different hosts, e.g. to support remote procedure calls among a network of loosely coupled hosts.

9. CONCLUSIONS AND LESSONS LEARNED

This successful implementation and demonstration of the ITCF proves that the ITCF concept is workable and that it has practical applications. Although the demonstration tools (especially the line editor) may not be heavily used in a real SEE, the same principles applied to adapt these tools could be applied to more popular screen editors and more sophisticated source reformatters and other source analysis tools.

The extension of a tool (such as an editor) through exploitation of the functionality of other tools certainly appears worth pursuing. Such extensions of functionality through integration of independent tools has several advantages. In the sample demonstration, tool functionality is made more accessible to the user. Without the use of ITCF integration, the user wishing to reformat his source program (under development) is forced to exit the editor, type a lengthy command to invoke the reformatter specifying the location of the program to be reformatted and the location for the resulting output (which may be subsequently discarded). The user must examine the results and then re-invoke the editor to continue development of the source program. Such obstacles to tool usage (in this case the use of the reformatter) are likely to leave such tools underutilized or even unutilized. On the other hand, with integration through the ITCF, the reformatting function becomes immediately available without exiting the editor. Only a simple command need be typed; the source is reformatted and immediately available for further inspection and editing.

Similarly, a user of the editor may wish to know if he has exceeded the source code development guidelines of his project (which require that Ada procedures contain less than 50 statements) as he is actually typing the procedure. If the user must exit the editor session in order to use the Ada statement counter tool, then the statement counter tool will be much less useful to him.

In addition to the user convenience which ITCF integration may provide, there are other advantages. Tools which are integrated through the ITCF are still independent of one another. They can be linked and maintained independently of one another. Separately linked tools integrated through ITCF occupy less memory than would be required if the tools were integrating by linking them into common memory images. Moreover, due to naming conflicts in independently developed tools, such linking is not always possible.

When a shared connection center with server tools is used by several users, each server tool is represented as a single, serially reusable process. This has several advantages. The tool images (even if non-reentrant) are represented only once in memory. The number of processes which must be managed by the underlying system is smaller. The tool images are not reloaded from disk each time they are invoked. The tool elaboration and other tool initialization is performed only once for all users.

A typical problem in shared processing environments is the use of large, resource-intensive tools (such as an Ada compiler) by several users concurrently. Sometimes as few as two concurrent invocations of such tools can drastically reduce system through-put due to contention for memory, processing, and IO resources. The ITCT can be used (as demonstrated) to conveniently serialize access to such tools in order to maximize through-put in these types of environments. For example, a common connection center could used by all CLI processes to serialize the use of the compiler and other tools

10. FUTURE APPLICATIONS

The connection center and other concepts employed in the ITCF are relatively new. The potential applications are as yet not fully defined. The design and implementation of the ITCF have been done with flexibility uppermost in order that the potential of this tool not be limited to specific integration scenarios that happened to be envisioned by the ITCF designers and implementers.

Other practical application scenarios should be investigated as the STARS program continues. There may be many as yet uninvestigated applications. One path toward wider application would be to make the ITCF facilities more readily available at the Command Line Interpreter level of the environment. This could be accomplished by implementing the ITCF mechanisms which support RPC (remote procedure call) and then integrating the ITCF functionality into ACE (the Ada Command Environment).

Another area for application of the ITCF could be application program development and testing. The use of software-first methodologies will lead to the need to test application software within the software development environment before the target hardware is available. Many applications to be developed are real-time systems utilizing tasks that respond to incoming "streams" of environmental data. There is clearly potential here to use the ITCF to create simulated environments and to monitor and record responses. There is even the possibility of direct use of ITCF stream management facilities in the applications themselves as a means of efficient inter-task communication and transient data management.

STARS Task BR-67
Inter-Tool Communication Facility (ITCF)
Final Report
13 February 1990

1. Introduction

The Inter-Tool Communication Facility (ITCF) was specified under STARS Task Q11. A VAX/CAIS-A implementation of the core ITCF facilities, a demonstration of the ITCF using existing tools, and this final report were completed under STARS Task BR67. This reports describes the ITCF, the design, the implementation, and the demonstration performed. More detailed documentation and information about the ITCF and the demonstration tools can be found in the ITCF Version Description and in the source code for the ITCF packages and for demonstration tools. All ITCF deliverables including source code and reports are currently on the MDSC VAX under the root directory:

```
   SYS$USER7[SOFTECH1.USERS.SABBUHL.ITCF]
```

The file:

```
   SYS$USER7[SOFTECH1.USERS.SABBUHL.ITCF]ITCF_VERSION_DESCRIPTION.ADS
```

describes the directory organization of the deliverables.

2. General Description

The following is a general description of the ITCF. More details can be found in the file ITCF_Version_Description.ads and in the package specifications and bodies for the delivered software.

The ITCF permits concurrently executing tools to cooperate in an integrated manner while maintaining their high degree of modularity and functional independence. The ITCF offers a layered architecture of communication services which may be adapted to meet the requirements of a variety of tool integration strategies. The principle conceptual layers are streams, stream protocols and messages.

The ITCF services manage the complex details of inter-tool communication and provide conceptually simple and easy to use interfaces to the tool writer. These services facilitate the integration of tools with relatively modest effort, including existing tools which were not originally designed to use the ITCF packages.

The ITCF is intended to transfer data which is primarily transient in nature as opposed to data which is permanently stored in the environment data base. Permanent data should be handled through the environment object management system. Nevertheless, simple tools can be implemented, using the ITCF packages, which allow the connection of other tools to files (including devices) as sources and sinks for inter-tool data.

The ITCF is also intended to support communication between tools executing in different CAIS-A environments that are connected by a gateway. However, this feature was not implemented under task BR67.

3. ITCF Concepts

The underlying concepts of the ITCF have been derived from a number of sources. These include UNIX pipes, plumbing fixtures and filters; the CAIS-A IO Connection Model; and various Remote Procedure Call (RPC) facilities and stream manipulation packages. The FIELD environment integration mechanism, described by Steven Reiss of Brown University, was particularly influential.

3.1 Clients and Connection Centers

The focus of ITCF operations are server processes called connection centers. A given connection center may have one or more clients. A client may be a tool (i.e. a process), or a gateway. A gateway client represents other clients and/or connection centers on a remote host system. Clients communicate with one another via the connection center.

The first step in using the ITCF is the creation of a connection center. After the center has been created, clients may be connected to the center. Connection of a client to a connection center establishes a communications channel between the client and the center. This channel is called a connection. Clients may be connected or disconnected at any time. Only clients of the same center may communicate with one another. Each client may be connected to only one center at a given time. Clients may communicate through the connection center concurrently.

3.2 Streams

At the lowest conceptual layer of the ITCF, data is passed through the connection center in the form of streams. A stream is simply a sequence of data (of indefinite length) generated by a client. A client sends data by writing into a stream. A client receives data by reading from a stream.

The flow of data in a stream is uni-directional. When bi-directional flow is required, a reverse stream called a reply stream may be used. A client process may use multiple streams concurrently. The flow of data in each stream is independent of other unrelated streams.

Streams may be used by processes for direct communication without the use of a connection center. Connection center facilities are built on top of the lower-level stream services. Streams may also be used locally within a process for communication between tasks without the overhead associated with sending streams through a separate connection center process.

4. Implementation Overview

The core facilities of the ITCF were implemented on the MDSC VAX machine using VAXAda 2.0 and Version 4.5 of the CAIS-A implementation. The ITCF services are provided as a collection of packages containing interfaces which may be called by CAIS tools.

The principal packages implemented are:

Stream_IO	- Supports all low-level stream operations including interprocess communication.
Text_Stream_IO	- Provides the interfaces of Ada Text_IO for reading and writing stream data.
Connection	- Provides interfaces that client tools call to interface with a connection center.
Connection_Center	- A generic package which supports the instantiation of user defined connection centers.
Distribution	- A generic package which performs general distribution list management functions for

	instantiations of package Connection_Center.
Monitor	- A low-level packages which is used to manage concurrent access to streams. This is an internal ITCF package which contains no tool interfaces.

5. Design Problems and Solutions

Aspects of the ITCF implementation which presented significant technical challenges were the management of concurrent access to streams and the interprocess connection resources, and the mechanism to support stream copying. Stream copying is a fundamental ITCF concept of substantial power in affecting the distribution of streams and the composition of streams from other streams.

5.1 Concurrent Access to Streams

Two problems needed to be addressed in the handling of concurrent access. The first was to develop a conceptual model that would make a correct implementation conceptually manageable. The second problem was to assure that management of concurrent access would be reasonably efficient.

5.1.1 Using Ada Rendezvous

Initial attempts to achieve a conceptual model for concurrent access were focused on the direct use of the Ada rendezvous model. One question was how tasking should be applied. The Ada rendezvous model is generally applied by identifying a object to which concurrent access may occur and then applying a task to manage the object. All access to the object is then accomplished through calling entries to that task. Hence, one attempt to the solve concurrent access problems in the ITCF was to have each stream be managed by a separate task. This approach was eventually rejected because it would require a large number of tasks, straining system resources. Also, it was not clear how stream copying could be implemented. Stream copying would either require 1) stream-tasks to call other stream-tasks, thereby making themselves unavailable for satisfying other concurrent access to the stream, or 2) the creation of addition copy-tasks that would simply read data from one stream-task and write-data to another. While such tasks could be recycled, thus reducing the overall overhead of frequent task creation, the number of tasks that would be required was still of some concern.

Another approach to using rendezvous was to define a single task which provides all services to all streams. This task would serialize access to all stream operations. While this approach would greatly reduce the number of tasks required, it had some disadvantages. One disadvantage was that all stream functions would have to be provided in a single large task body. In order to modularize these functions, calls would have to be used in the ACCEPT statements of this task body adding additional overhead (which could possibly be reduced through the use of PRAGMA INLINE). Also, how to interface with other tasks which manage transmission of streams over connections was not clear.

While these problems probably could be overcome, there was one problem that was shared by all approaches that used Ada rendezvous exclusively: every operation on a stream would involve four tasks switches. Task switching has certain unavoidable overhead even in the best Ada runtime implementations. All registers must be stored and loaded, and the scheduling algorithm must be invoked. This would imply that an operation which is to write a single byte to a stream could have an overhead on the order of a hundred instructions. This overhead could be significantly reduced by using buffering to minimize the number of rendezvous required to transmit a given amount of data. However, this would only help in the Put and Get functions in package Stream_IO. The overhead for all other functions would not be reduced.

5.1.2 Using Monitors

An alternative approach was to apply Hoare's Monitor concept. In this approach, the set of all streams is considered a shared resource to which access is control via a monitor entity. Whenever a task needs to operate on a stream, it must first enter the monitor. Monitor entry is queued so that only one task can execute within the monitor at a time. If the operation cannot be completed immediately (for example an attempt to read a stream which is temporarily empty), the task may suspend itself on a condition variable until the operation can be completed. When a task suspends itself within a monitor, another task is allowed to enter. Eventually, a task will enter which writes to the stream and signals the condition variable upon which the suspended task is waiting.

5.1.3 Monitor Implementation

The monitor framework was very helpful in sorting out the concurrent access problems. However, since Ada does not directly support monitors, it was necessary to consider how they would be implemented. Waiting must occur when a task attempts to enter a monitor, if another task is using the monitor, and when a task must be suspended on a condition variable. In Ada, the only way for a task to wait until some event occurs is through a call to a task entry. A very simple and direct approach to the monitor entry problem is to use a simple task which alternately accepts an entry call to enter the monitor and an entry call to leave the monitor. The major drawback to this solution is that the cycle of entering and leaving the monitor would require two rendezvous involving eight task switches.

However, an important observation is that, in most cases, when a task attempts to enter the monitor there will be no task executing in the monitor. Thus, in most cases a rendezvous should be unnecessary. In the ITCF implementation, a counter which can be incremented/decremented and tested as an atomic operation is used to determine whether a task is currently executing within the monitor. A rendezvous is used to suspend an entering task only if the result of incrementing the counter indicates that another task is already executing inside the monitor. This strategy reduces the overhead of monitor entry and exit to a few instructions, versus the hundreds of instructions that would be executed using rendezvous exclusively.

The technique used could be criticized for not being pure Ada. However, the only implementation dependency introduced by this technique is the implementation of the atomic counter on the host. Most processors have such an instruction. Others have an instruction (such as a test and set) which can be used to synthesize an atomic counter. In the case of the VAX, the VAXAda compiler has a built-in procedure for an atomic counter. For other hosts or compiling systems, it may be necessary to write a small assembly language unit.

In any case, the dependency is isolated within a small package called Protected_Counter. There is no dependency on the Ada run-time implementation. All tasking operations are still handled in pure Ada.

5.2 Stream Copying

Several false starts were required in the design of stream copying before a satisfactory solution was arrived at. The final solution was to place an item representing the "tail" (i.e. the remainder) of a copied stream in the item queue of every target stream to which the stream is copied. These tails are kept on a list. Each time another data buffer is added to the source stream, an item referencing that buffer is inserted into each target stream immediately ahead of each tail (in the list of tails) for the copied stream. This effectively copies the data into each target stream at the appropriate point within each of those streams. When the copied stream is closed, the tails are discarded since no further insertions of data from the copied stream are necessary.

In applications involving large streams, the amount of data which is buffered in a stream must be bounded to prevent storage resources from being exhausted. Handling of bounded buffering in streams was problematic due to the copying facility. Initial, it was not clear how to account for buffered data which has been copied. As the design of the copy capability progressed, it was decided that buffers would be copied by reference rather than by value for efficiency reasons. The accounting problem of placing bounds on buffering was then solved by designating the stream to which data is originally written as the owner of that data. Thus, copying a stream A to a stream B has no affect on the amount of data that can be written directly to stream B. A shared buffer is only released after every reference to it in any stream has been deleted. Thus, write operations to stream A which has been copied to stream B may be blocked until data is read from both streams A and stream B.

6. Changes to the Original Design

Few changes were made to the ITCF specifications during the implementation. However, one significant change was a redistribution of the functionality provided by the packages Stream_IO and Stream_Services. During the implementation it was realized that the underlying stream services (which were not detailed in the specification) could be useful outside the context of ITCF connection centers. Thus, interfaces originally intended to be hidden in the implementation of Stream_Services were instead made a visible part of package Stream_IO. Thus, package Stream_IO now provides full access to low-level stream functionality. Package Stream_Services was eliminated.

Higher-level functionality, associated specifically with the use of a connection center, was removed from package Stream_IO and placed in package Connection which already included interfaces through which client processes communicate with a connection center. (One regret here is the package name Connection which would more appropriately be Center_IO. This could be fixed in a future version.) The rearrangement of these services improves ITCF modularity since general-purpose stream services (provided by Stream_IO) can now be used without the overhead of also binding-in connection center support.

7. Demonstration of the ITCF

The selection of a demonstration for the ITCF was a compromise between the desire to demonstrate a meaningful, non-trivial ITCF application and the need to devote most of the available time to the implementation of the ITCF. A meaningful demonstration would be one in which the functionality of the existing tools is integrated. The repository containing STARS and SIMTEL tools was considered as the primary source of existing tools. In order to minimize the risk of demonstration problems, attention was focused on relatively simple, small tools. A demonstration of the ability to integrate tools was of more interest than the tools themselves. For example, integration of the ITCF with ACE (Ada Command Environment) was considered but was not undertaken, primarily because ACE is large body of software which would have had to be ported from a UNIX environment to CAIS-A on a VAX. Also, the integration of ITCF with ACE would not in itself demonstrate the utility of the ITCF. At least two more tools would be required to perform some sort of demonstration.

7.1 Tool Selection

There were several tools in the repository which could perform operations on Ada source code. These included editors, spelling checkers, Ada statement counters and Ada source code reformatters. Since programmer's spend much of their time using text editors, it seemed that a natural choice for a demonstration would be one which extended the functionality of a text editor through integration with other text processing tools. Thus, a demonstration was conceived in which an editor would be the primary tool interfacing with the user and would be integrated with other text processing tools operating as servers through a connection center. This scenario seemed to provide a relatively straight-forward demonstration.

A simple line editor, written in Ada, was chosen from the repository. Although it is unlikely that this particular editor would be a heavily used tool in an actual SEE, the same types of extensions could be made to the more popular varieties of screen editors. For purposes of the demonstration, additional commands were added to the editor which would be carried out through a connection center. In particular, new commands to reformat and to count statements in Ada source code being edited were added. These new commands cause the contents of the current editor buffer to be transmitted to other tools via a connection center and a reply to be received by the editor.

An Ada source code reformatter was selected from the repository. There were at least two choices. The simpler and smaller reformatter was chosen. Initially, a spelling checker was also selected as a server tool for the demonstration. However, as time began to run out it was decided to replace the spelling checker with a simpler tool, an Ada statement counter, to reduce the risk of not meeting schedules. The main reasons for the substitution was that the spelling checking had a very long startup time and its user interface seemed complex. It was not clear how much effort would be required to adapt it for an acceptable demonstration.

In order to demonstrate the broadcast and reply concatenation capabilities of the ITCF, another simple command was added to the line editor. This command causes a message to be sent to all clients of a connection center which have registered themselves as server tools. Each tool responds with its registered name. These responses are concatenated into a single stream by the connection center and delivered back to the editor which prints the response. Hence, the net affect of the this new editor command is to poll for the names of all available server tools which are connected to the center.

7.2 Tool Adaptation

Adaptation of the tools chosen for integration via the ITCF was straight-forward. In the case of the line editor, the new commands were readily added. However, the command for reformatting suggested that another editor extension would be necessary. The problem is that if there are mistakes in the source code, the results of reformatting could be erroneous in which case the user may wish to restore the prior contents of the editor buffer in place of the reformatted contents. The mechanism used by the editor to store the text being edited was easily modified to handle two text buffers instead of one. Another command was added to the editor which allows the user to toggle between these buffers. Thus, after a buffer is formatted the user has the option of switching back to the original text buffer as it was before invoking the reformatter.

Both the reformatter and the Ada statement counter tools were easily adapted since they already used TEXT_IO for purposes of input and output. Adaptation of these tools was largely a matter of adding a few calls to the ITCF package Connection. Each tool had to perform the following initialization steps:

1. Call Open_Connection to initialize the connection to the connection center.

2. Call Register_Pattern to register the identification of the tool with the connection center.

3. Call Activate_Connection to notify the connection center that the tool was ready to receive request streams.

Thereafter, the tools would call the Receive interface in package Connection to receive each stream containing source code to be processed. The tools would then open each stream using TEXT_STREAM_IO. Since these tools already used TEXT_IO to read their input and write their output, the TEXT_IO calls had only to be changed to call TEXT_STREAM_IO instead.

More detailed information about the changes made to tools for adaptation can be found in the Ada source code files for the tools.

7.3 Tool Adaptation Problems

Although the changes necessary to integrate the tools with the ITCF were minor, other aspects of tool behavior became important in the context of integration. This was particularly evident in the reformatter tool. The reformatter, as it was written, would stop processing if it encountered anything which is not correct Ada in its input stream. While this sort of behavior may be acceptable in a stand alone tool, in an integration setting it seemed counter-productive. If the user sent his editor buffer to the reformatter, only part would come back in the (very likely) event that the source code contained errors. Hence, changes were made to the reformatter to cause it to pass remaining source code through unchanged in the event of a syntactic error from which it could not recover.

The reuse of tools such as the reformatter in this way increases the need for tool reliability. While occasional failures of a stand-alone tool may be tolerable, the failure of a tool in a closely integrated scenario may not be so well tolerated. For example, an obvious and implied extension of the demonstration scenario would be the multiplexed use of the connection center and the server tools (the reformatter and the statement counter) by several independent invocations of the editor tool. That is, the services provided by the connection center could be provided to multiple independent users. In this case, if one user sends an source program to the reformatter which causes it to abort, then the facility is also lost to the other users of the connection center.

Hence, the robustness of tools becomes more critical. It may be that future adaptations of tools to the ITCF should include considerations such as restarting a tool if it fails. At the very least, when one tool fails, other tools connected to connection center should not necessarily fail as a consequence.

7.4 Tool Adaptation Effort

Much of the effort that was necessary to adapt the reformatter was spent in trying to fix existing weaknesses and make it more robust. There are still existing bugs in this tool which could not be fully investigated in the time available. Approximately a two man-weeks of effort were applied to this tool.

Adaptation of the Ada statement counter was trivial, requiring only a few hours. It was only necessary to add a main processing loop.

Adaptation of the line editor was fairly straight-forward. Some knowledge of its internal structure was necessary in order to add the new commands and to add the capability of multiple buffers which could be swapped via a command. The editor adaptations required approximately a man-week.

Note that all adaptations were performed by a programmer familiar with the ITCF interfaces but completely unfamiliar with the tools being integrated. No consultation from others was sought in performing the adaptations. Realistically, this situation (no consultation) probably reflects typical future tool adaptation scenarios. However, those with expertise in these tools could have performed the necessary adaptations more quickly.

7.5 Instantiation of the Demonstration Connection Center

For flexibility, ITCF connection center software is provided in the form of generic packages which must be instantiated by the user to create a connection center having the desired distribution strategy. The connection center instantiations used in the demonstration are in a package called Integration_Center. The actual parameters for the instantiation include an Ada type defining patterns and a matching function which compares streams to be distributed against patterns. The Integration_Center package defines patterns to be simple strings up to 20 characters long. These strings are the names of the server tools connected to the Integration_Center, e.g. "Formatter" and "Ada Count". Requests sent by the line editor to the connection center have a header which indicates which tool is to receive the stream. A stream with the header "*" is sent to all of the server tools connected. The "*" header is used to implement the editor command which polls the server tools connected to the connection center.

The implementation of the Integration_Center was relatively simple since most of the work is already taken care of in the of bodies of the generic packages. However, instantiation of package Distribution is non-trivial, requiring a detailed understanding of how to distribute call streams and how to concatenate their corresponding reply streams. The process of making a connection center could be made substantially easier by providing a higher-level generic package which would automatically instantiate the package Distribution with a standard distribution strategy.

7.6 Demonstration Performance Results

Responsiveness of the special editor commands was found to be fairly good once all tools and the connection center were established and initialized. Typical response times for small editor buffers is one or two seconds. However, the initial establishment of the connection center and the tools is fairly lengthy primarily due to the many node creation operations perform by the CAIS-A implementation in creating the processes and IO channels (six nodes are created for the demonstration configuration). It is expected that these creation times will be much shorter in future, improved releases of the CAIS-A implementation.

Performance was significantly poorer when large streams were transmitted. A three page source file can take on the order of 20 seconds to be reformatted. The computational overhead for stream management is relatively low. The apparent cause of the performance problem with large streams is the use of polling in the low-level IO mechanism. It is likely that performance would be much better if this polling mechanism were replaced with an interrupt-driven IO mechanism.

7.7 Running the Demonstration

The following steps may be used to run the demonstration on the MDSC VAX:

1. Logon as user SOFTECH1, password GREENE.

2. Type: RUN UOBJ:CAIS_LOGON to logon to the CAIS.
   Specify user ITCF_DEMO, password ITCF_DEMO.

3. Wait for the prompt "file?". This may take a minute.

4. Type the name of a VMS file to read into the buffer
   or
6. Type a carriage return to start an empty buffer.

7. Type "H" (help) and "S" (summary) for an editor command summary.

8. Type "A" (append) to enter an Ada program from the keyboard.
   Type "." CR to end input mode.

9. Type "L" (list) "A" (all) to list the editor buffer contents.

10. Type "C" (center) to poll the tools connected to the ITCF connection center. The names of the tools will be printed.

11. Type "Z" to send the editor buffer to the Formatter tool.
    A new buffer will be returned.

12. Type "B" (buffer switch) to toggle back to the unformatted buffer.

13. Type "K" to send the editor buffer to the Ada statement counter. The number of Ada statements, blank lines, comment lines etc. will be printed.

14. If CONTROL-Y must be typed, refresh the database before logging into the CAIS again, as follows:

    Type CONTROL-Y
    Type STOP
    Type @REFRESH

8. Unimplemented Features

Not all features of the ITCF were implemented under this task. Only the core facilities required for the demonstration were fully implemented. In particular, the following ITCF packages were not implemented:

Package Binary_Stream_IO	- protocol for transmitting Ada data structures in a compact binary form.
Package Sequential_Stream_IO	- an ITCF version of Ada's Sequential_IO.
Package Message_IO	- a generic package for the convenient encoding and decoding of short streams.

Also, the capability to connect together several connection centers was not implemented. This feature is useful primarily in the implementation connections between different hosts, e.g. to support remote procedure calls among a network of loosely coupled hosts.

9. Conclusions and Lessons Learned

This successful implementation and demonstration of the ITCF proves that the ITCF concept is workable and that it has practical applications. Although the demonstration tools (especially the line editor) may not be heavily used in a real SEE, the same principles applied to adapt these tools could be applied to more popular screen editors and more sophisticated source reformatters and other source analysis tools.

The extension of a tool (such as an editor) through exploitation of the functionality of other tools certainly appears worth pursuing. Such extensions of functionality through integration of independent tools has several advantages. In the sample demonstration, tool functionality is made more accessible to the user. Without the use of ITCF integration, the user wishing to reformat his source program (under development) is forced to exit the editor, type a lengthy command to invoke the reformatter specifying the location of the program to be reformatted and the location for the resulting output (which may be subsequently discarded). The user must examine the results and then re-invoke the editor to continue development of the source program. Such obstacles to tool usage (in this case the use of the reformatter) are likely to leave such tools underutilized or even unutilized. On the other hand, with integration through the ITCF, the reformatting function becomes immediately available without exiting the editor. Only a simple command need be typed; the source is reformatted and immediately available for further inspection and editing.

Similarly, a user of the editor may wish to know if he has exceeded the source code development guidelines of his project (which require that Ada procedures contain less than 50 statements) as he is actually typing the procedure. If the user must exit the editor session in order to use the Ada statement counter tool, then the statement counter tool will be much less useful to him.

In addition to the user convenience which ITCF integration may provide, there are other advantages. Tools which are integrated through the ITCF are still independent of one another. They can be linked and maintained independently of one another. Separately linked tools integrated through ITCF occupy less memory than would be required if the tools were integrating by linking them into common memory images. Moreover, due to naming conflicts in independently developed tools, such linking is not always possible.

When a shared connection center with server tools is used by several users, each server tool is represented as a single, serially reusable process. This has several advantages. The tool images (even if non-reentrant) are represented only once in memory. The number of processes which must be managed by the underlying system is smaller. The tool images are not reloaded from disk each time they are invoked. The tool elaboration and other tool initialization is performed only once for all users.

A typical problem in shared processing environments is the use of large, resource-intensive tools (such as an Ada compiler) by several users concurrently. Sometimes as few as two concurrent invocations of such tools can drastically reduce system through-put due to contention for memory, processing, and IO resources. The ITCF can be used (as demonstrated) to conveniently serialize access to such tools in order to maximize through-put in these types of environments. For example, a common connection center could used by all CLI processes to serialize the use of the compiler and other tools.

10. Future Applications

The connection center and other concepts employed in the ITCF are relatively new. The potential applications are as yet not fully defined. The design and implementation of the ITCF have been done with flexibility uppermost in order that the potential of this tool not be limited to specific integration scenarios that happened to be envisioned by the ITCF designers and implementers.

Other practical application scenarios should be investigated as the STARS program continues. There may be many as yet uninvestigated applications. One path toward wider application would be to make the ITCF facilities more readily available at the Command Line Interpreter level of the environment.

This could be accomplished by implementing the ITCF mechanisms which support RPC (remote procedure call) and then integrating the ITCF functionality into ACE (the Ada Command Environment).

Another area for application of the ITCF could be application program development and testing. The use of software-first methodologies will lead to the need to test application software within the software development environment before the target hardware is available. Many applications to be developed are real-time systems utilizing tasks that respond to incoming "streams" of environmental data. There is clearly potential here to use the ITCF to create simulated environments and to monitor and record responses. There is even the possibility of direct use of ITCF stream management facilities in the applications themselves as a means of efficient inter-task communication and transient data management.

STARS Task BR-67
Inter-Tool Communication Facility (ITCF)
Demonstration
13 February 1990

The demonstration tools, tool adaption, problems and results are all documented in Section 7 of the ITCF Final Report. Demonstration software can be found in directories under the following root directory on the MDSC VAX:

```
SYS$USER7[SOFTECH1.USERS.SABBUHL.ITCF]
```